.NET Framework

based on:

A. Troelsen, Pro C# 2005 and .NET 2.0 Platform, 3rd Ed., 2005, Apress
J. Richter, Applied .NET Frameworks Programming, 2002, MS Press
D. Watkins et al., Programming in the .NET Environment, 2002, Addison Wesley
T. Thai, H. Lam, .NET Framework Essentials, 2001, O'Reilly
D. Beyer, C# COM+ Programming, M&T Books, 2001, chapter 1

Krzysztof Mossakowski
Faculty of Mathematics and Information Science
http://www.mini.pw.edu.pl/~mossakow

Contents

- The most important features of .NET

- Assemblies
- Metadata
- Common Type System
- Common Intermediate Language
- Common Language Runtime
- Deploying .NET Runtime

- Garbage Collection
- Serialization

.NET Benefits

In comparison with previous Microsoft's technologies:

- Consistent programming model – common OO programming model
- Simplified programming model – no error codes, GUIDs, IUnknown, etc.
- Run once, run always – no "DLL hell"
- Simplified deployment – easy to use installation projects
- Wide platform reach
- Programming language integration
- Simplified code reuse
- Automatic memory management (garbage collection)
- Type-safe verification
- Rich debugging support – CLR debugging, language independent
- Consistent method failure paradigm – exceptions
- Security – code access security
- Interoperability – using existing COM components, calling Win32 functions

Base Class Libraries

- Classes available to all .NET Framework languages
- Various primitives:
  - threads, file input/output, graphical rendering, interaction with external hardware devices
  - database access, XML manipulation, security

.NET Programming Languages

- http://www.dotnetpowered.com/languages.aspx
- Examples:
  - C#
  - Managed Extensions for C++
  - Java - Visual J# .NET
  - JavaScript - JScript .NET
  - Perl
  - Pascal, Delphi
  - PHP
  - Smalltalk

.NET Assemblies

- Binaries containing Common Intermediate Language (CIL) instructions and type metadata
  - .dll or .exe files, which cannot be run without the .NET runtime
- The most important features:
  - establishing a type boundary
  - versioning
  - self-describing
  - configurable

.NET Assembly's Format

- .NET assembly consists of the following elements:
  - Win32 File Header
  - CLR File Header
  - CIL code
  - type metadata
  - assembly manifest
  - optional embedded resource

Single-File and Multifile Assemblies

- In a great number of cases, there is a simple one-to-one correspondence between a .NET assembly and the binary file (**`.dll`** or **`.exe`**)
  - this is single-file assembly
- Multifile assemblies are composed of numerous .NET binaries (modules)
  - one of these modules (primary module) must contain the assembly manifest
  - multifile assemblies allow to use more flexible deployment option (e.g. the user is forced to download only selected modules)

Private Assemblies

- Private assemblies are required to be located in application's directory or subdirectory
- Identification of a private assembly:
  - name of the module that contains the assembly's manifest (without an extension)
  - version number
- Probing – the process of mapping an external assembly request to the location of the requested binary file

Shared Assemblies

- Single copy of a shared assembly can be used by several applications on a single machine
- A shared assembly should be installed into the Global Assembly Cache (GAC), located in **`Assembly`** subdirectory of Windows directory
  - since VS 2005 also **`*.exe`** files can be installed into the GAC (previously only **`*.dll`** files were accepted)
- To list content of the GAC, install a new assembly, or uninstall an assembly, use **`gacutil.exe`** utility
  - installing an assembly:
    `gacutil.exe –i MyAssembly.dll`
- Two or more assemblies of the same name can coexist in the GAC (must have different versions)
  - the end of "DLL hell"

Signing an Assembly

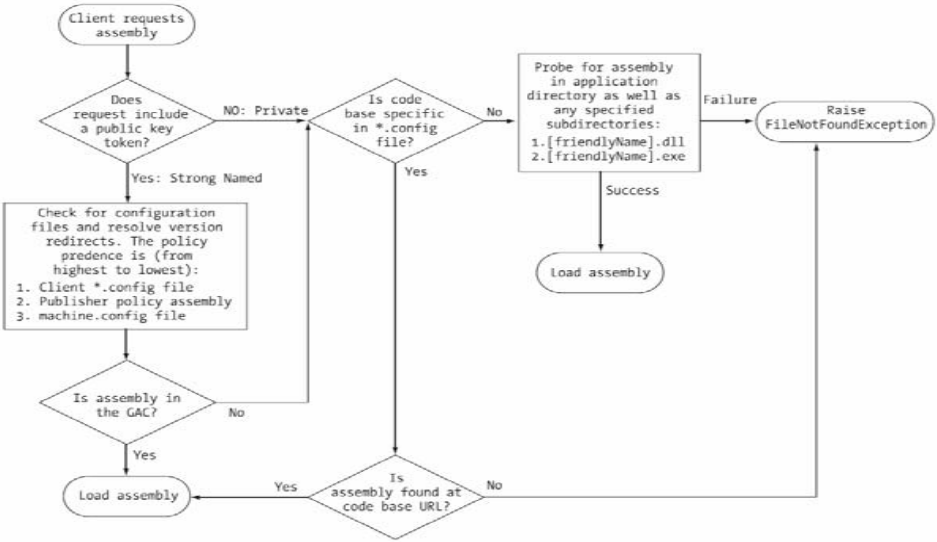
- Only assemblies signed using a strong name can be installed into the GAC
- Signing an assembly:
  1. Use the **`sn.exe`** utility to generate a **`.snk`** file with public/private key information
     `sn -k MyKey.snk`
  2. Apply the **`.snk`** file to the assembly (using **`AssemblyKeyFile`** attribute or by setting project's properties in Visual Studio)
  3. Compile the assembly

Configuring an Assembly

- Assemblies can be configured using **`*.config`** files
  - simple XML files that can be manually edited or configured using .NET Framework 2.0 Configuration utility (**`mscorcfg.exe`**)

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="CarLibrary"
                          publicKeyToken="219ef380c9348a38"
                          culture=""/>
        <bindingRedirect oldVersion= "1.0.0.0"
                         newVersion= "2.0.0.0"/>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

Resolving an External Assembly Reference



Assembly Manifest

- Manifest is a piece of metadata which describes the assembly itself
- Manifest documents all external assemblies required by the current assembly to function correctly
- Content of a manifest:
  - the name of the assembly
  - the version of the assembly
  - the shared name for the assembly
  - information about the type of environment the assembly supports (e.g. operating system and languages)
  - list of files in the assembly
  - list of all other assemblies this assembly references

.NET Type Metadata

- .NET assembly contains full, complete and accurate metadata, which describes:
  - all types (classes, structures, enumerations)
  - all members of types (methods, properties, events etc.)
- Metadata is emitted by a compiler
- Some benefits of using metadata:
  - no need to register in a system (unlike COM objects)
  - no need for header files
  - tips from IntelliSense in Visual Studio
  - crucial for some .NET technologies, e.g. remoting, reflection, late binding, XML web services, and object serialization
  - garbage collection

Reflection

- Reflection is a process of runtime type discovery
- Allows to programmatically obtain metadata information

- **`System.Reflection`** namespace:
  - **`Assembly`**, **`AssemblyName`**, **`EventInfo`**, **`FieldInfo`**, **`MemberInfo`**, **`MethodInfo`**, **`Module`**, **`ParameterInfo`**, **`PropertyInfo`**
- **`System.Type`** class

Attributes

- A way for programmers to embed additional metadata into an assembly
  - attributes are code annotations that can be applied to a given type, member, assembly, or module
- .NET attributes are class types that extend the abstract **`System.Attribute`** base class
- Some predefined attributes:
  - **`[CLSCompliant], [DllImport], [Obsolete], [Serializable], [NonSerializable], [WebMethod]`**
- Custom attributes can be created

Common Type System (CTS)

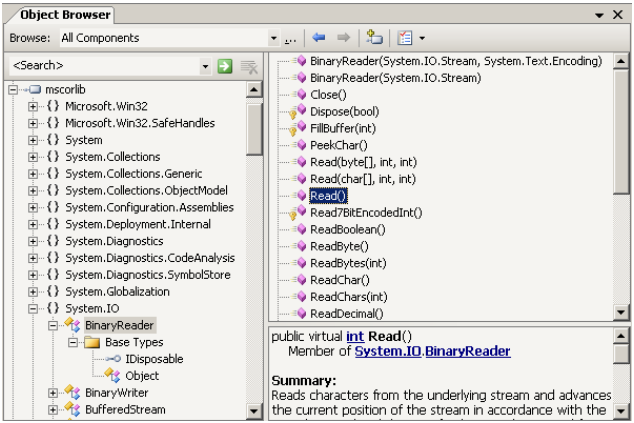
- Types in .NET
  - classes (sealed classes, implementing interfaces, abstract classes, internal or public classes)
  - structures
  - interfaces (named collections of abstract member definitions)
  - enumerations
  - delegates (equivalent of type-safe function pointer)
- CTS is a formal specification that documents how types must be defined in order to be hosted by the CLR

Intrinsic CTS Data Types

CTS Data Type	VB.NET	C#	Managed C++
System.Byte	Byte	byte	unsigned char
System.SByte	SByte	sbyte	signed char
System.Int16	Short	short	short
System.Int32	Integer	int	int or long
System.Int64	Long	long	__int64
System.UInt16	UShort	ushort	unsigned short
System.UInt32	UInteger	uint	unsigned int or unsigned long
System.UInt64	ULong	ulong	unsigned __int64
System.Single	Single	float	Float
System.Double	Double	double	Double
System.Object	Object	object	Object^
System.Char	Char	char	wchar_t
System.String	String	string	String^
System.Decimal	Decimal	decimal	Decimal
System.Boolean	Boolean	bool	Bool

Type Distinction - Namespaces

- Namespace is a grouping of related types contained in an assembly
- A single assembly can contain any number of namespaces



Standard .NET Namespaces

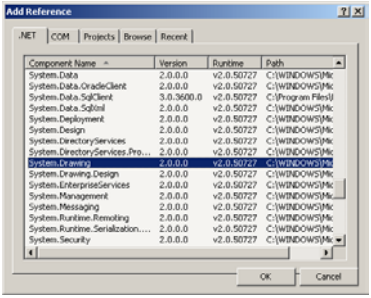
System	Types dealing with intrinsic data, mathematical computations, random number generation, garbage collection, exceptions, attributes
System.Collections System.Collections.Generic	Stock container objects, base types and interfaces used for building customized collections; generics
System.Data System.Data.Odbc System.Data.OracleClient System.Data.OleDb System.Data.SqlClient	ADO.NET for database solutions
System.Diagnostics	Source code debugging and tracing
System.Drawing System.Drawing.Drawing2D System.Drawing.Printing	Types wrapping graphical primitives such as bitmaps, fonts, and icons; printing capabilities

Standard .NET Namespaces – cont.

System.IO System.IO.Compression System.IO.Ports	File I/O, buffering, compression, serial ports
System.Net	Network programming, sockets
System.Reflection System.Reflection.Emit	Runtime type discovery, dynamic creation of types
System.Runtime.InteropServices	Interaction with unmanaged code (DLL and COM)
System.Threading	Support for multithreaded applications
System.Web	ASP.NET, XML Web Services
System.Windows.Forms	Windows Forms (GUI for Windows applications)
System.Xml	Interaction with XML data

Referencing External Assemblies

- To use types from external assembly:
  1. Add a reference to the project



  2. Use fully qualified names

```
System.Drawing.Bitmap bmp =
                    new System.Drawing.Bitmap(50, 50);
```

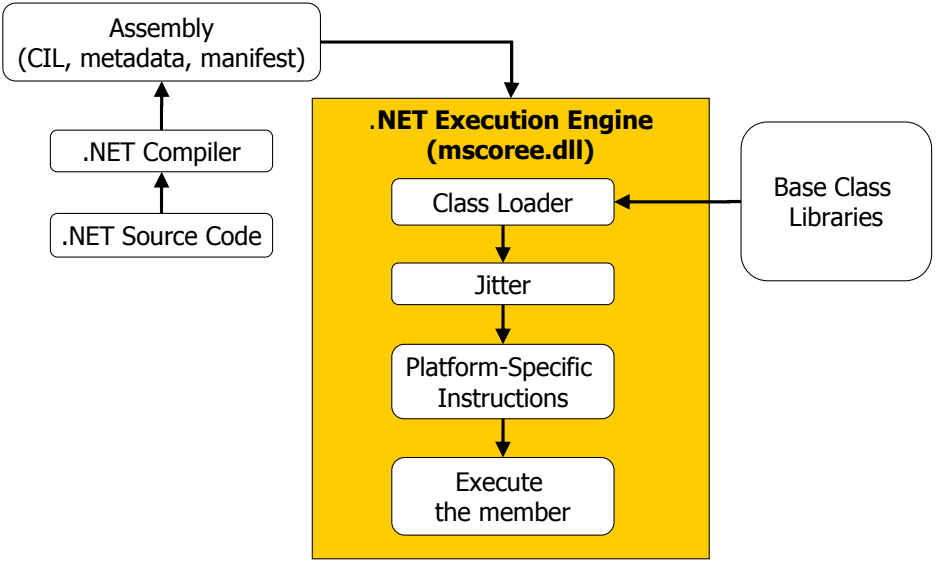
or utilize **`using`** directive

```
using System.Drawing;
```

Common Intermediate Language (CIL)

- Also known as Microsoft Intermediate Language (MSIL)
- CIL is a language that sits above any particular platform-specific instruction set
  - the same idea as Java's virtual machine
- Compilers of all .NET-aware languages emit CIL instructions
  - binaries are platform-independent
- When the CIL code is about to run, the Jitter (just-in-time compiler) compiles it into native (machine) code
  - Jitter will cache resulting machine code in memory

CLR Workflow



Reverse Engineering

- CIL makes reverse engineering of any .NET solution very easy
  - metadata
- Tools:
  - IL Disassembler (ildasm.exe), included in .NET Framework SDK shows CIL code
  - Lutz Roeder's .NET Reflector (http://www.aisto.com/roeder/dotnet/) shows CIL code and its representation in C#, VB.NET, C++, Delphi, and other languages

Obfuscating the CIL

- The purpose of an obfuscator: to modify .NET assembly without affecting its functioning, to make it difficult or impossible to recover source code
- Potential downsides of obfuscating:
  - can break code that depends on reflection, serialization, or remoting
  - can make diagnosing and debugging problems
  - adds another step to build process
- Obfuscation in Visual Studio .NET
  - Community Edition of Dotfuscator for .NET supports basic entity renaming and removal of unused metadata
- Commercial obfuscators

Obfuscation Methods

- Entity renaming
  - changing names of namespaces, classes, methods, properties, fields, enumerations
- Control flow obfuscation
  - modifying the original code (e.g. transforming **`if`** or **`while`** statements by using **`goto`** statement)
- Removal of unused members
- String encryption
- Breaking IL Disassembler
  - injecting code into the obfuscated assembly that is designed to break IL Disassembler so that it won't open the assembly at all
- Compiling into native code

Common Language Specification (CLS)

- CLS is a set of rules provided to:
  - describe the minimal and complete set of features to produce code that can be hosted by CLR
  - ensure that products of compilers will work properly in .NET environment
- Sample rules:
  - representation of text strings
  - internal representation of enumerations
  - definition of static members

Common Language Runtime (CLR)

- CLR is physically represented by **`mscoree.dll`** library (Common Object Runtime Execution Engine)
  - this library is loaded automatically when an assembly is referenced for use
- CLR responsibilities:
  - resolving the location of an assembly and finding the requested type within the binary by reading the contained metadata
  - loading the type into memory
  - compiling CIL into platform-specific instructions
  - performing security checks
  - executing the code

Deploying .NET Runtime

- .NET assemblies can be executed only on a machine that has the .NET Framework installed
- .NET Framework 1.1 was included in Windows Server 2003 and was an optional component of Windows XP Service Pack 1
- Redistributable packages (dotnetfx.exe)
  - .NET Framework 1.1
    - 23.1 MB
    - Windows 98/Me/NT/2000/XP/2003, IE 5.01
  - .NET Framework 2.0
    - 22.4 MB
    - Windows 98/Me/NT/2000/XP SP2/2003, IE 5.01
    - Windows Installer 3.0
    - disk space: 280 MB (x86), 610 MB (x64)

Mono and .NET Portable

- Mono project (http://www.mono-project.com)
  - support for .NET client and server applications on Linux, Solaris, Mac OS X, Windows and Unix
  - sponsored by Novell
  - LGPL (Lesser General Public Licence)
- Portable .NET (http://www.dotgnu.org)
  - supported systems: GNU/Linux, NetBSD, FreeBSD, Cygwin/Mingw32, Mac OS X, Solaris, AIX
  - GPL (General Public Licence)
- Official international standards:
  - ECMA-334: The C# Language Specification
  - ECMA-335: The Common Language Infrastructure (CLI)

.NET Framework 2.0 SDK

- .NET Framework 2.0 SDK
  - requires Windows 2000 SP3/XP SP2/2003
  - requires .NET Framework Redistributable Package installation
  - contains command line tools, e.g. csc.exe – C# compiler and cordbg.exe – debugger
  - occupies 354 MB
  - there are Language Packs which contain translated text, such as error messages
    - Polish – 1.9 MB

IDE Tools for C# and .NET

- Visual Studio 2005
  - versions: Express, Standard, Professional, Team, Tools for Office
  - .NET Framework 2.0 SDK included
- SharpDevelop (http://www.sharpdevelop.com)
  - Windows
  - LGPL
- Mono Develop (http://www.monodevelop.com)
  - Linux, Mac OS X
  - GPL

Garbage Collection (GC)

- .NET objects are allocated onto a region of memory termed the managed heap
- They will be destroyed by the garbage collector
  - make no assumption about time of destruction
- Garbage collection (an attempt to free up memory) will be performed when CLR determines that the managed heap does not have sufficient available memory
  - all active threads are suspended
  - special GC thread tries to free memory
  - suspended threads are waken up
- GC in .NET is highly optimized

Garbage Collection Process

- The runtime investigates objects on the managed heap to determine if they are still reachable
  - object graph is built
- Unreachable objects are marked as garbage for termination and swept from memory
- The remaining space on the heap is compacted

- To optimize the process, two distinct heaps are used:
  - one is specifically used to store very large objects and is less frequently consulted during the collection cycle

Object Generations

- To help optimize the process, each object on the heap is assigned to a specific generation
  - the idea: the longer an object has existed on the heap, the more likely it is to stay there
- Used generations:
  - Generation 0: newly allocated objects that have never been marked for collection
  - Generation 1: objects that survived one sweep
  - Generation 2: objects that have survived more than one sweep
- Garbage collector starts from generation 0 objects, if not enough memory was released, it works with generation 1 objects, and later with generation 2

System.GC Class

- **`AddMemoryPressure(), RemoveMemoryPressure()`** – changes settings of the GC of the need of memory
- **`Collect()`** – forces to perform garbage collection
- **`CollectionCount()`** – returns a value representing how many times a given generation has been swept
- **`GetGeneration()`** – returns the generation to which an object currently belongs
- **`GetTotalMemory()`** – returns the estimated amount of memory (in bytes) currently allocated on the managed heap
- **`MaxGeneration`** – the maximum of supported generations (2 for NET 2.0)
- **`SupressFinalize()`** – sets a flag indicating that the specified object should not have its **`Finalize()`** method called
- **`WaitForPendingFinalizers()`** – suspends the current thread until all finalizable objects have been finalized

Forcing a Garbage Collection

- Sample scenarios when forcing a garbage collection can be useful:
  - the application is about to enter a block of code which should not be interrupted
  - the application has just finished allocating an extremely large number of objects and as much memory as possible should be freed

```
GC.Collect();
GC.WaitForPendingFinalizers();
```

Finalizable Objects

- **`Finalize()`** method is declared as a destructor in C# and C++ languages
- Garbage collector will call an object's **`Finalize()`** method (if supported) before removing the object from memory

- Important recommendation: design classes to avoid supporting **`Finalize()`** method
  - time of calling this method is unpredictable

Disposable Objects

```
public class MyClass : IDisposable
{
    public void Dispose()
    {
        // here dispose all memory
        // - all unmanaged resources
        // - call Dispose() method of all contained
        //   disposable objects
    }
}

using (MyClass mc = new MyClass()) {
    // ... using mc object
}   // here Dispose() method is called automatically
```

Formalized Disposal Pattern

```
public class MyResourceWrapper : IDisposable
{
    private bool disposed = false;
    public void Dispose() {
        CleanUp(true);
        GC.SuppressFinalize(this);
    }
    private void CleanUp(bool disposing) {
        if (!disposed) {
            if (disposing) {
                // dispose managed resources
            }
            // clean up unmanaged resources
        }
        disposed = true;
    }
    ~MyResourceWrapper() {
        CleanUp(false);
    }
}
```

Serialization

- Serialization is a process of persisting (and possibly transferring) the state of an object to a stream
- The persisted data sequence contains all necessary information needed to reconstruct (deserialize) the state of the object
- When an object is persisted to a stream, all associated data (base classed, contained objects, etc.) are automatically serialized as well
- It allows to persist an object graph in a variety of formats
- Full set of related objects (so-called object graph) is serialized

Serialization Attributes

- All objects in an object graph to serialize must be marked with the **[Serializable]** attribute
  - all public and private fields of a class marked with this attribute are serializable by default
- **[OptionalField]** attribute can be used for fields that can be missing

```
[Serializable]
public class MyClass
{
    public bool boolToSerialize;
    private int[] arrayOfIntsToSerialize;

    [NonSerialized]
    public string notToSerialize;
}
```

Serialization Formatters

Formatters available in .NET 2.0:

- **`BinaryFormatter`** – compact binary format
- **`SoapFormatter`** – SOAP message
- **`XmlFormatter`** – XML document

- Only **`BinaryFormatter`** preserves full type fidelity (each type's fully qualified name and the full name of the assembly is stored)

Serialization Sample

```
MyClass mc = new MyClass();
//SoapFormatter formatter = new SoapFormatter()
BinaryFormatter formatter = new BinaryFormatter();

// serializing
Stream stream = new FileStream("out.dat",
                               FileMode.Create,
                               FileAccess.Write,
                               FileShare.None);
formatter.Serialize(stream, mc);
stream.Close();

// deserializing
stream = File.OpenRead("out.dat");
MyClass mc2 = (MyClass)formatter.Deserialize(stream);
stream.Close();
```

XML Serialization

```
MyClass mc = new MyClass();
XmlSerializer formatter = new XmlSerializer(
       typeof(MyClass),
       new Type[] {typeof(MyClass)});

// serializing
Stream stream = new FileStream("out.xml",
                               FileMode.Create,
                               FileAccess.Write,
                               FileShare.None);
formatter.Serialize(stream, mc);
stream.Close();

// deserializing
stream = File.OpenRead("out.xml");
MyClass mc2 = (MyClass)formatter.Deserialize(stream);
stream.Close();
```

Customizing the Serialization Process

- .NET Framework 1.1
  - implement **`ISerializable`** interface
- .NET Framework 2.0
  - use attributes:
    `OnDeserializedAttribute`
    `OnDeserializingAttribute`
    `OnSerializedAttribute`
    `OnSerializingAttribute`
    `OptionalFieldAttribute`
- **`SerializationInfo`** object is a "property bag" that maintains name/value pairs representing the state of an object during the serialization process